

AD-A285 790



RL-TR-94-162  
Final Technical Report  
September 1994



①

# INCREMENTAL REDERIVATION OF SOFTWARE ARTIFACTS: FY93 FINAL REPORT

The MITRE Corporation

Melissa Chase, Howard Reubenstein, and Alexander Yeh

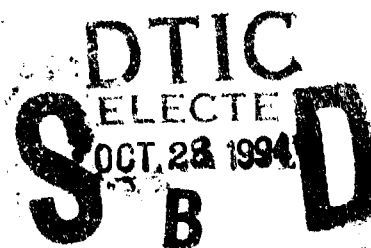
DTIC QUALITY ASSURANCE

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

94-33547



268



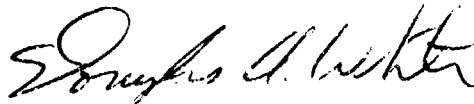
Rome Laboratory  
Air Force Materiel Command  
Griffiss Air Force Base, New York

94 10 27 006

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-94-162 has been reviewed and is approved for publication.

APPROVED:



DOUGLAS A. WHITE  
Project Engineer

FOR THE COMMANDER:



JOHN A. GRANIERO  
Chief Scientist  
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL ( C3CA ) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE September 1994		3. REPORT TYPE AND DATES COVERED Final Oct 92 - Sep 93
4. TITLE AND SUBTITLE INCREMENTAL DERIVATION OF SOFTWARE ARTIFACTS: FY93 FINAL REPORT			5. FUNDING NUMBERS C - F19628-91-C-0001 PE - 62702F PR - 5581 TA - 27 WU - 57	
6. AUTHOR(S)  Melissa P. Chase, Howard Reubenstein, and Alexander Yeh			8. PERFORMING ORGANIZATION REPORT NUMBER  N/A	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The MITRE Corporation 202 Burlington Road Bedford MA 07130			10. SPONSORING/MONITORING AGENCY REPORT NUMBER  RL-TR-94-162	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3CA) 525 Brooks Rd Griffiss AFB NY 13441-4504				
11. SUPPLEMENTARY NOTES  Rome Laboratory Project Engineer: Douglas A. White/C3CA/(315) 330-3564				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Design replay presents a possible enabling technology to the Knowledge-Based Software Assistant specification maintenance and implementation derivation approach to software development. More generally, design replay can also be applied to derivations between a variety of different software description abstraction levels. Radical changes to a software artifact cannot generally be addressed by design replay as they require new design output. Evolutionary changes are more amenable to design replay and often involve incremental changes to derived artifacts. This report describes an approach to rederivation that exploits the incrementality of evolutionary maintenance changes, the state of MITRE's implementation of this approach, and what remains to be done to test this approach.				
14. SUBJECT TERMS Knowledge-based software engineering, Software engineering, Automatic programming, Formal specifications, (see reverse)			15. NUMBER OF PAGES 28	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

14. (Cont'd)

Artificial intelligence, Evolutionary development, Program derivation, Replay,  
Rederivation



# Incremental Rederivation of Software Artifacts: FY93 Final Report

Melissa P. Chase, Howard Reubenstein and Alexander Yeh

The MITRE Corporation

202 Burlington Road

Bedford, MA 01730

[pc,hbr,asy]@mitre.org

June 27, 1994

## Abstract

Design replay presents a possible enabling technology to the Knowledge-Based Software Assistant's specification maintenance and implementation rederivation approach to software development. More generally, design replay can also be applied to derivations between a variety of different software description abstraction levels. Radical changes to a software artifact cannot generally be addressed by design replay as they require new design input. Evolutionary changes are more amenable to design replay and often involve incremental changes to derived artifacts. This report describes: an approach to rederivation that exploits the incrementality of evolutionary maintenance changes, the state of our implementation, and what remains to be done to test this approach.

## 1 Introduction

The problem we are exploring is the insertion of a design replay capability into Rome Lab's Knowledge-Based Software Assistant (KBSA) [1]. The KBSA is composed of a number of facets that roughly correspond to activities in a standard waterfall life-cycle model. These facets include requirements, specification, and development. Note that the KBSA does not force a waterfall development; in fact it is based on a new paradigm of software development involving maintenance of specifications and rederivation of implementations. Each facet assists in a design activity producing increasingly formal descriptions of a desired system. Each facet can be viewed as taking a transformational design approach.

Our task is constrained by two important requirements. First, we wish to provide a design replay utility that can be used by any and all of the facets. Thus, we are taking an approach reminiscent of the Joshua system's [2] "protocol of inference." We are interested in designing a protocol that a design tool can use to record the: decisions it is making, alternatives it is considering, goal structure of its problem solving, and data dependencies

between actions it takes and artifacts created. This protocol should permit capture of the information necessary to permit application of incremental redesign techniques.

Second, we want to be able to use incremental replay from the *current* state of the code artifact to reconstruct the code for a modified design. (In this context, the use of the word code should be qualified as referring to the output artifact of a particular facet. This artifact may be source code, but it could be specification, requirements or other descriptions.) We do not want a strategy that requires replay of an entire design.

To explain an important difference between software design problems and other areas amenable to replay technology, planning for example, particularly with respect to this second requirement, an analogy to the blocks world is helpful. In blocks world planning there are two artifacts of particular concern: the plan to achieve the desired goal and the actual configuration of the blocks (before and) after executing the plan. In software, the corresponding artifacts are the design (plan) to produce the code and the code itself (block configuration). Incremental replanning techniques [3] assist in the creation of a new plan/design, but not in the formation of a new block configuration/code from the state left by the original plan. This latter capability is critical in software construction where the design has already been applied to yield code.

Maintenance must proceed from this existing code for a number of reasons. The design process is likely to have been quite long, e.g., 10,000 transformations [4] for a medium-sized problem, and on average might only be 90% automated. Therefore, replay of an entire design would require user assistance in remaking 10% of the design decisions, e.g., 1,000 decisions, many of which may not even have been affected. Even if the process were 99% automated, 100 decisions would still have to be remade by the user. Also, depending on the efficiency of the transformation system, even fully automatic replay of 10,000 transformations may be something to avoid. In essence, there are two incremental modification problems to be considered simultaneously in software design, that of modifying the design and that of modifying the source code.

The next section describes our approach to design replay. Following this are sections describing related work and the state of our implementation.

## 2 Incremental Rederivation

A major complication in design replay is the "correspondence problem" [5, 6]: establishing a correspondence between the components of the original initial specification and the changed initial specification. This problem is both difficult and necessary to solve for replay to occur: the original set of transformations need to be replayed on the components of the changed initial specification that correspond to the components of the original initial specification which the set of transformations were originally applied to. An interesting aspect of the incremental rederivation approach is that it finesses this correspondence problem to a large extent. The input to the replay capability consists of: an original initial specification, a transformational development and its associated dependency information as described below, a solution artifact, and a delta to the original specification. The required output is a new solution artifact that reuses as much of the initial solution as possible and minimizes its interaction with the underlying design performance system/agent. Correspondence is not a

significant issue under this problem definition because the change is localized to an area of the original specification and the dependency structure identifies and propagates the other correspondences in the transformational development.

To explore this further, we need to look at an initial simple definition for transformations and a basic artifact representation. The transformation representation we are concerned with is derivative from [7] and is meant to capture only the simplest syntactic properties of a transformation. Similarly, the artifact representation, which must be facet independent to a large extent, captures only basic syntactic properties.

Artifacts will be represented as abstract syntax trees (AST). This provides a level of representation above the purely textual but without facet specific semantic concepts.<sup>1</sup> Transformations, at this initial level of description, consist of an input pattern that matches pieces of the base AST and an output modification to the AST. The section of the AST to be modified must be included in the input pattern. The parts of the AST matched by the input pattern form the *input span* of the transformation and the modified sections comprise the *output span*.

## 2.1 Example

An example of an AST for the LISP file containing

```
(defun fact (x) (:input))
dummy
```

is shown in Figure 1.

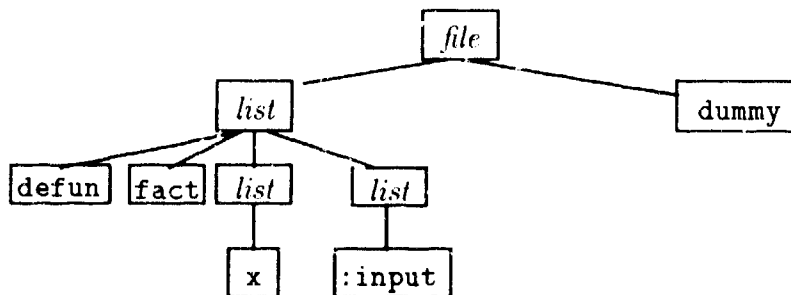


Figure 1: AST for (defun fact (x) (:input)) dummy

An example transformation rule is:

rule fact-input (input, name, arg)

input = '(:input \$existing-cond)' & name = 'fact' -->

input = '(:input (integer @(copy(arg))) (>= @(copy(arg)) 0) \$existing-cond)'

The rule's name is *fact-input*. The AST node that it may modify is called *input* in the rule, and the other AST nodes that it examines are called *name* and *arg*. The rule states that if

<sup>1</sup> Facet specific semantic concepts can be helpful in rederivations but are excluded for now because their use will vary from one facet and system to another.

1. *input* corresponds to an expression of the form `(:input  $\alpha$ )`, where  $\alpha$  is zero or more arbitrary expressions
2. and *name* corresponds to an expression of the form `fact`

then *input* gets changed to an AST corresponding to an expression of the form

$$(:input (integer\ arg_{\beta}) (>= arg_{\gamma} 0) \alpha)$$

where  $arg_{\beta}$  and  $arg_{\gamma}$  are copies of the AST bound by *arg* (the function *copy* makes a copy of its argument). This rule's input span is the union of the AST pieces bound by the variables *input*, *name* and *arg*. The rule's output span is the AST piece that *input* gets transformed into.

An example transformation is applying the above rule to the example AST so that the rule's *input* parameter is bound to the part of the AST for `(:input)`, *name* is bound to the part for `fact`, and *arg* is bound to the part for `x`. These bound parts form the input span. The transformation alters the AST for `(:input)` to an AST for `(:input (integer x) (>= x 0))`. This latter piece forms the output span. So with this transformation, the original AST is transformed into the AST for

$$(\text{defun fact (x) } (:input (integer\ x) (>= x 0))) \\ \text{dummy}$$

## 2.2 Representing Design Rationale

A factor in choosing an algorithm or data structure is often whether some condition is true, e.g., "data access occurs at least ten times more frequently than data updates" or "the number of data elements will not exceed one hundred." With the above representation of artifacts and transformations, such conditions can be represented as specification "comments" (of a specific syntactic form) within the implementation. Then, any transformation that depends on such a condition can have one of the transformation's input parameter variables matched against a pattern of the appropriate form. For example, the second condition instantiated for data of type *brand-x* could be represented with a specification comment of the form

$$(:condition-of-world (maximum-number\ brand-x\ 100))$$

Then a transformation rule could test for the presence of such a condition by including the following in its applicability tests:

$$\text{numelements} = '(:condition-of-world (maximum-number\ @type\ @maxnum))' \\ \& (<= \text{maxnum}\ 100)$$

where *numelements* is a rule input parameter to be bound to the specification comment describing the condition, *type* is a rule input parameter to be bound to the type of data element, and *maxnum* is a rule variable.



## 2.3 Macro-Level Rederivation

Given an input specification  $S0$ , a solution artifact  $Sf$ , and a transformational development history with input and output span information preserved, changes to the input specification can be propagated at a macro-level as follows: given<sup>2</sup> a change to a piece of  $S0$  we can compute a partition of the transformations into two sets:

1. those transformations whose input span has definitely not been altered and so are definitely unaffected and can therefore be reused intact, and
2. those transformations which are possibly affected because their input span may have been affected. The input span may be affected if it includes part of the piece of  $S0$  that has been altered or it includes part of an output span of another transformation that may have been affected.

This partition leads to a partition of  $Sf$ , where each component is either

1. part of the output span of a transformation that is definitely unchanged or is directly a piece of  $S0$  that has remained unchanged, or is
2. part of the output span of a transformation that may be affected or is directly a piece of  $S0$  that has been changed.

The first type of  $Sf$  component is definitely unaltered and can be used as is, while the second type may have been altered.

This macro-level rederivation is a form of impact analysis that in itself can be useful in large systems under the assumption that many of the maintenance changes made to a specification only have relatively local effects in the solution artifact. This assumption is restrictive but consistent with the typical maintenance profile where the difficult problem is reliably identifying the relatively small percentage of the code that needs to be changed to respond to a maintenance request.

An example of macro-level rederivation is as follows. Start with the example transformation (that adds input conditions) described in Section 2.1. Let "dummy" in the original AST be altered to be "useless". Because **dummy** was not part of the input span for the transformation, that transformation is *not* affected by the alteration and can be used as is. As a result, the transformation's output span, (:input (integer x) (>= x 0)) in the transformed AST, is definitely unaltered and can be used as is. Also, the rest of the transformed AST:

(defun fact (x)

comes directly from parts of the original AST that have *not* been altered and so is also definitely not affected by the alteration. However, since "dummy" in the transformed AST comes

---

<sup>2</sup>These various inputs are provided by a design tool interacting with the replay component through the defined information and dependency protocol. This protocol defines the data a design tool must record about its ongoing design process in order to make use of the replay facilities. The protocol will not be discussed further in this paper.

directly from part of the original AST that has been altered, that part of the transformed AST has been altered.

On the other hand, if  $x$  in the original AST, which is part of the input span for the transformation, were altered to be  $y$ , then the transformation would possibly be affected and the transformation's output span, `(:input (integer x) (>= x 0))` in the transformed AST, would also possibly be altered.

Given the set of possibly affected transformations, we can attempt to replay the transformational history to rederive new solution components. For each transformation that we might consider replaying, however, there is a spectrum of possibilities from redo (and possible reinvocation of the design system/agent) to extracting finer-grained transformation dependencies that permit more precise control of reinvocation of a transformation. This finer level of dependency information is discussed in the next section.

Two goals in keeping this dependency information should be recalled here. First, we have the goal of essentially performing impact analysis through the dependency structure. This analysis gains us time efficiency by both not replaying or examining in detail entire huge derivations (which is traded off in space required for the dependencies) and by not interacting with a person for manual information extraction. Second, we wish to be able to closely analyze those transformations that are possibly affected by new input requirements. Many transformations will be unaffected and therefore not require replay. Those that are affected and still applicable will often be modifiable through the dependency structure. If a transformation is no longer applicable and the design program needs to be reinvoked, then it is likely that rederivation of the entire subsolution will be necessary. If an appropriate impact analysis is supported, then this will be more efficient as fewer valid solution components will be needlessly rederived.

## 2.4 Micro-Level Rederivation

Before discussing finer-grained dependency information, we need to refine our representation of transformations. We have stated that transformations consist of an input pattern and an output modification. Following Balzer in [7], a transformation also consists of a set of local bindings (used to compute partial substructures) and a set of constraints to be verified. All data accessed in the constraints, bindings, or output modification must be accessed by the input pattern, i.e., the input span of a transformation must identify all data dependencies.

An input pattern implicitly embodies two kinds of constraints, i.e., equality to a constant and equality between substructures (use of the same named pattern variable).

Bindings can perform arbitrary computation but we make a few assumptions about them. First, we assume all data accessed by the binding computation is functionally indicated in the binding calculation call. This implies that a binding calculation is functional, i.e., a calculation always returns the same result on the same arguments and does not access internal state. Second, we assume none of the bindings are dead/unused. We could attempt to compute this and, in fact, dead bindings should not affect the ultimate propagation through the dependency structure.

Constraints may also perform arbitrary computation, perhaps accessing a reasoning system to check properties of the input pattern. As with bindings, we assume all these compu-

tations are also functional.

Given this finer-grained transformation representation, it is now possible to either compute the fact that a delta to a transformation's input pattern actually has no effect on the validity of the output modification or, more likely, that the output modification can be updated via an incremental recomputation not requiring reinvocation of the transformation. To understand how this will work, we must look at the kinds of changes that can occur in the input span of a transformation and how they propagate through the transformation.

#### 2.4.1 Replay Delta Calculus

To enumerate the space of deltas that can affect a transformation, we shall step through the computation of whether a transformation is still applicable. A delta to a piece of an AST is postulated and the macro-level dependency mechanism identifies a transformation as having an input span that is possibly affected by the delta. First, the input pattern must be matched against the changed AST and the delta localized to the affected portions of the input pattern.<sup>3</sup>

If the change in the input pattern is in a constant portion of the pattern, then the transformation is no longer applicable. If the transformation is to a variable portion of the pattern then we continue examining the change. Next, we compute a delta set on the local transformation variable bindings using the delta set of the input pattern. Then we compute a possible delta set for the constraints (and any equality constraints implicit in the input pattern) based on the delta sets for bindings and variables. For each possibly changed constraint, we reevaluate its truth. If any constraint is false, the transformation is no longer applicable. If all the constraints are still true, we move on to computation of the new output pattern.

The output pattern of a transformation may not be changed if all deltas simply affect triggering conditions. The output pattern is changed if it incorporates any of the changed inputs or bindings. At this point, we could simply recompute the new output pattern by recomputing the output modification of the transformation. However, in some cases it is possible to propagate the deltas through the output pattern instead of performing a full recompute. For example, two possible action types in an output pattern include: creating a constant subcomponent (this is unaffected by deltas in the input) or copying a variable binding in as a subcomponent. This has the advantage of not requiring reinvocation of the transformation and permitting localization of the deltas to parts of the output pattern.

Two examples of this checking and propagation process are as follows. Again, start with the example transformation described in Section 2.1.

1. If **fact** in the original AST were altered to **not-fact**, then the corresponding binding to the transformation rule's *name* variable would also be altered to **not-fact**. Since the rule checks that *name* matches the constant pattern **fact** as a condition of its applicability, this alteration would render the transformation inapplicable.
2. If instead, **(:input)** in the original AST were then to be altered to **(:input z y)**, then the corresponding binding to the transformation rule's *input* variable would also

---

<sup>3</sup>The original match of the input pattern to the AST could be kept as part of the dependency structure and then a delta to a piece of the AST could be mapped directly to the affected part of the input pattern.

be altered to be (:input *z y*). Then the rule would find that this would match against the pattern (:input \$existing-cond) when the variable *existing-cond* is bound to the list (*z y*) (*existing-cond* was originally bound to a list of zero elements). Since the rule has no constraints on *existing-cond*'s binding, the rule would still be applicable. If, however, the rule had a constraint that *existing-cond* be bound to a list of less than two elements, then the rule would no longer be applicable.

Given that the rule is still applicable, one will then look at how the altered bindings for *input* and *existing-cond* may affect the output. *Input* is not used to produce the output and is "ignored." *Existing-cond*'s binding on the other hand is copied into the output after ( $\geq \gamma 0$ ), where  $\gamma$  is *x*, so the transformation's output would be updated with this new binding to:

(:input (integer *x*) ( $\geq x 0$ ) *z y*)

The original output was (:input (integer *x*) ( $\geq x 0$ )).

In the case where a change occurs to a pattern that appears multiple times in the input, it may be the case that there is an unintentional violation of an implicit program design equality constraint between the multiple subcomponent occurrences. Before continuing with replay, we may want to consult the user to verify the intentions. A simple example of such a pattern (for the C language) is

for (@*x* = 0; @*y* < *N*; @*z*++) @statement

where the variables (marked with a @) *x*, *y* and *z* are explicitly constrained to bind to references to the same variable, as in for (*i* = 0; *i* < *N*; *i*++) *a*[*i*]++;. This pattern is of an iterative construct where the index variable (*i* in the example) is initialized to 0, the iterations continue as long as the index variable is < *N*, and the index variable is incremented at the end of each iteration. If a user altered

for (*i* = 0; *i* < *N*; *i*++) *a*[*i*]++;    into    for (*k* = 0; *i* < *N*; *i*++) *a*[*i*]++;

so that *x* now refers to a different variable from *y* and *z*, one might wonder if the user was just trying to rename *i* with *k* and forgot to update all the occurrences of *i*.

In the case where a change occurs in multiple output subcomponents versus a single subcomponent, we have an indication of where a change has an expanding impact.

The output deltas identified from one stage of propagation now form the input deltas for the next stage of propagation which continues until all affected transformations are processed.

## 2.5 A Larger Example

The best example of how these techniques work would involve a large derivation and a typical maintenance change affecting only a small part of the implementation artifact. For example, in a database access system consisting of data storage and retrieval routines, data indexes, report generators, and other functions, a user might make a change to an output specification that simply resulted in the alteration of a few constants in the report generation

routines. For illustrative purposes, we will consider a smaller example that is an expanded version of the example given in Section 2.1.

The input specification for this example is:

```
(DEFUN FACT (X)
  (:DESIGN RECURSIVE) (:DESIGN CACHE-RESULTS) (:DESIGN INDUCTIVE)
  (:INPUT) (:OUTPUT) (:IMPLEMENTATION) (:COMMENT))
```

This input is transformed into an output artifact by the application of eight transformation rules. The first rule is the *fact-input* rule described in Section 2.1. The other rules are written in a similar fashion. The details are given in Appendix A. Applying these rules transforms the specification into the result shown in Figure 2. In the results, the (LIS?) code

```
(DEFUN FACT (X)
  (:DESIGN RECURSIVE) (:DESIGN CACHE-RESULTS) (:DESIGN INDUCTIVE)
  (.INPUT (INTEGER X) (>= X 0))
  (:OUTPUT (INTEGER (FACT X)))
  (:IMPLEMENTATION
    (LET ((RET-7375
      (COND ((FACT-VAL-CACHED?-7375 X) (FACT-CACHED-VAL-7375 X))
            ((NOT (AND (INTEGER X) (>= X 0))) :ERROR)
            ((> X 0) (* X (FACT (- X 1))))
            ((= X 0) 1))))
      (CACHE-FACT-VAL-7375 X RET-7375)
      RET-7375))
  (:COMMENT))

(DEFVAR *FACT-CACHE*-7375 (MAKE-HASH-TABLE) "cache for FACT")

(DEFUN FACT-VAL-CACHED?-7375 (X) "check to see if value cached for FACT"
  (GETHASH X *FACT-CACHE*-7375))

(DEFUN FACT-CACHED-VAL-7375 (X) "return cached value for FACT"
  (GETHASH X *FACT-CACHE*-7375))

(DEFUN CACHE-FACT-VAL-7375 (X VAL) "cache a value for FACT"
  (SETF (GETHASH X *FACT-CACHE*-7375) VAL))
```

Figure 2: Result of Applying the Eight Transformation Rules

for the main FACTorial function is given in the part marked by :IMPLEMENTATION, and the code for the associated functions and variable for caching is given in the DEFUNs and DEFVAR.

Suppose that one wants to delete (:COMMENT) from the original specification. How much of the transformational history will need to be replayed? First, looking at the macro-level,

it turns out that the input span for each of the first five transformations is limited to one or more of the following parts (or transformations of those parts): FACT, X, (:DESIGN RECURSIVE), (:DESIGN INDUCTIVE), (:INPUT), (:OUTPUT), (:IMPLEMENTATION). So (:COMMENT) is *not* part of the input span of these transformations and the transformations can be reused as is from the original transformation development. However, transformation 6 (see Appendix A) needs the entire development produced so far in order to be able to attach the caching variable and functions to the development. So (:COMMENT) is part of transformation 6's input span and one will have to go to the micro-level to see how much of the transformation can be reused. Transformations 7 and 8 use part of transformation 6's output span for their input spans, so they too will need examining at the micro-level. As a result, using macro-level rederivation, the first five transformations can be reused without alteration or detailed examination.

Next, we look at the effects of deleting (:COMMENT) from the original specification on transformation 6 at the micro-level. It turns out that all transformation 6 does with (:COMMENT) is copy it, without any checking, from 6's input span to one location in 6's output span. So updating transformation 6 does not require replaying the entire transformation, but just deleting (:COMMENT) from the output span. After this update, examining transformation 7's input span reveals that it does not include the altered part of transformation 6's output span, so transformation 7 can be reused as is from the original transformation development. The same holds true of transformation 8. The final result of this rederivation process is the same as the original result (Figure 2) with (:COMMENT) removed.

In this example, incremental rederivation saved one from having to replay any of the eight transformations after a minor change in the original specification. Five transformations could be reused as is after just a cursory examination. Two more transformations could be reused as is after a more detailed examination. Just one transformation needed alteration, and this alteration was a minor one which was analogous to the change made in the original specification. For larger derivation histories, or those where individual transformations are time consuming or ones with manual interactions, this incremental strategy will produce even larger gains in savings in the amount of work required to obtain a new artifact.

## 2.6 The Dependency Structure

Each transformation in a derivation sequence causes a delta to part of the software artifact under construction. The AST for an artifact consists of a set of attributes and keywords organized according to the syntax of the relevant artifact source language.

In recording the transformational development, two views of the artifact are maintained: the original base artifact and the current transformed version. The current version is simply a caching of the results of tracing the transformational development from the original artifact through to the current time. The derivation history is captured as a set of extra-linguistic annotations on the AST rooted initially at the original AST. These annotations capture transformation applications as a tuple of [ input-span, output-span, transform, time-stamp ] (and necessary cross-references). The input-span and output-span are identified as sub-trees in the AST. The derivation history annotations include copies of the output-span results of each transformation. The current transformation dependency structure can be traversed by starting at the original base artifact and following the transformation annotations through

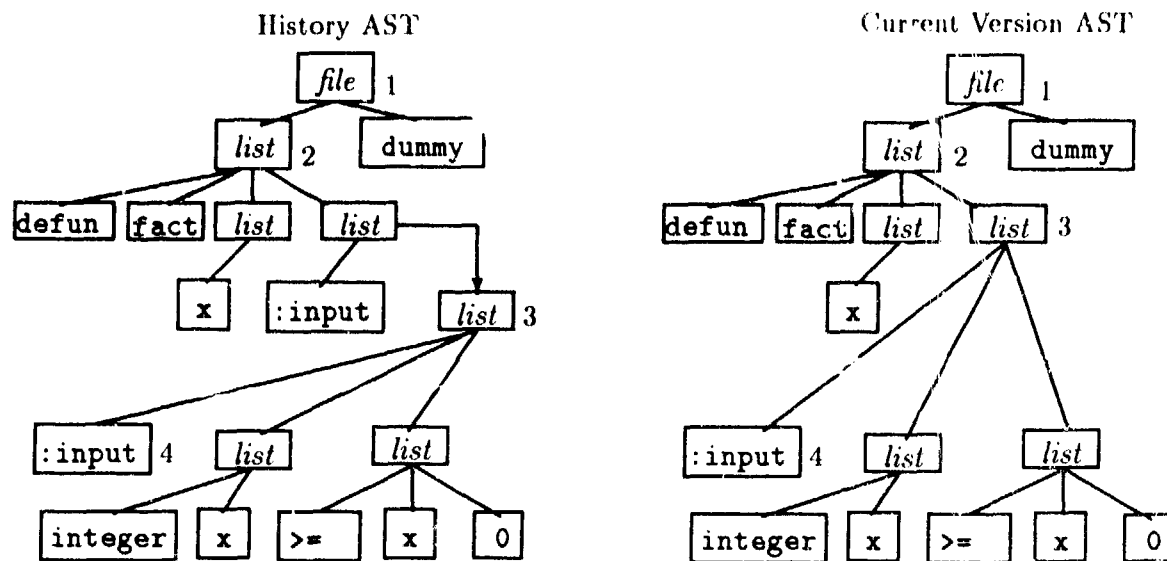


Figure 3: AST's for History and Current Version after applying a transformation

the structure as ordered by the time-stamp attribute.

An example of AST's that record a transformational history and display the current version are partially shown in Figures 1 and 3. Figure 1 shows the ASTs that record the transformational history and display the current version (the two look the same) before the "fact-input" transformation described in Section 2.1 takes effect. Figure 3 shows the AST's after the "fact-input" transformation takes effect. As before, the lines *without* arrowheads show the sub-expression/parent-expression relationships among the nodes. The line *with* an arrowhead shows the rule's transformation of (:input) into

(:input (integer x) (>= x 0)).

The numbers 1 through 4 indicate the correspondences between some of the nodes (to the left of the number) in the history AST and the nodes in the current version AST.

The history AST permits tracing through the impacts of the transformational development process. The current version AST caches the result of the process up to the current time. Using this dependency structure, the rederivation process can occur after a change to the original specification/artifact as follows:

1. mark which parts of the history might be altered by changing part of the original artifact
2. find the version of the artifact that is produced as a result of applying all the unaffected transformations (macro-level rederivation)
3. find the possibly affected transformations
4. perform micro-level rederivation on the possibly affected transformations for which micro-level rederivation is known to be sufficient to update the transformation

5. replay the possibly affected transformations which micro-level rederivation may not be able to handle

Currently, all steps except for micro-level rederivation and the generation of new artifact fragments for altering an input specification by insertion have been implemented and tested. The examples (outside of the parts involving micro-level rederivation) have been run successfully on the implementation. Details of the current implementation can be found in [8].

Storing a dependency structure will take an amount of memory proportional to how large the input and output components of individual transformations are and the actual number of applied transformations. It is clear that this structure can grow quite large. It can also stay of manageable size, e.g., imagine a 10,000 line program and a 10,000 transformation derivation each of which touches on average 1/10% of the code, 10 lines. This will result in an overhead proportional to 10 times program size.<sup>4</sup> It is clear, however, that the dependency structure will continue to grow as the design process proceeds and thus strategies for compressing this structure must be considered. Use of these strategies will be driven by experiments regarding the actual memory overhead.

A drastic compression strategy would involve discarding the current start state  $S_0$  and creating a new start state from some intermediate point in the derivation. Essentially this would discard the ability to roll back a derivation to the original input artifact state. A more moderate strategy would involve keeping the dependency trail for macro-level rederivation, but eliding information necessary for micro-level recomputation. A compromise strategy would involve computing the transitive closure of a sequence of transformations and thus compress the information about that sequence into a single virtual transformation.

### 3 Related Work

Concentrating on rederivation in maintenance applications is an interesting simplification to adopt since it finesses the correspondence problem that occurs in other reuse strategies such as applications of derivational analogy. Adopting a dependency-directed impact analysis approach obtains the benefits of typical serial replay approaches without incurring the overhead of manual requerying and dead-end derivation adaptation while trading the serial replay time cost for a memory cost.

The replay work in the KIDS system [5] emphasizes a derivational analogy approach [10] to reuse concentrating on the correspondence problem. This approach is particularly useful in reusing a previous solution to solve a new but similar (analogical) problem. In our work, we are assuming the problem is more than similar. In fact, we are assuming that the problem is substantially identical except for a defined delta corresponding to a maintenance update.

Baxter takes an approach to reusing lengthy design histories [4] that involves propagating maintenance deltas through the design history to obtain a reordering of the history into a prefix sequence that is unaffected by the maintenance delta (or at least reusable) and the balance of the history that is no longer appropriate. The reusable prefix is then rerun and problem solving proceeds from that point. This is done taking the goal structure of

---

<sup>4</sup>This is the same size as the estimate given by Neighbors for DRACO[9].



the design history into account. We take a maintenance delta approach, like Baxter, but propagate a delta through a transformation dependency structure instead of the design history. The advantage of our approach is that it just needs to trace through dependencies at a syntactic level, while Baxter's approach will, in general, need proofs involving program semantics. Such proofs will involve general theorem proving and/or enumerations of large numbers of possibilities, both of which are hard to perform. One way to view our work is that it is finding useful cases of reuse where checks on the semantics can actually be done by performing simple tests at the syntactic level. PRIAR [11, 12] also takes into account the dependency structure of a derivation, in this case a plan, to find a maximally reusable plan. As in Baxter's work, the unnecessary parts of a reused plan are removed and new parts are added to establish unmet goals. Other approaches for reusing maximal parts of a design history and then reinvoking a performance program include the REMAID [13] experiments.

Feather's work on parallel elaboration via evolution transformations and merging [14, 15, 16] points to a transformational development methodology that should yield highly replayable derivation histories. Elaboration of parallel design considerations allows non-interfering parts of a development to emerge separately. Where the parallel paths need to be merged, the merge process will identify dependencies. Feather assumes a serial replay process as one mechanism to achieve merging. Our dependency-directed approach does not seem initially applicable to achieving merge via replay since the nature of the merge process is to resolve undocumented dependencies. However, the dependency-driven approach should be highly effective on the resulting replayable derivation histories.

## 4 Status and Conclusions

We have defined the requirements for our replay capability and have designed the supporting representations and some of the algorithms. As was mentioned in Section 2.6, except for micro-level rederivation and some types of input specification alterations, an implementation of our approach to rederivation has been built and tested. If we continue implementing, we would limit micro-level rederivations to ones that either just determine the applicability of a transform or propagate the same alteration to the output of a transformation as was made to the input of that transformation. This limitation will make implementing micro-level rederivations more tractable while providing enough power to handle many of the micro-level rederivations that one will want to perform, including all the micro-level rederivations needed for the examples.

The incremental dependency-directed rederivation approach we have described is an excellent match to maintenance type artifact modifications in which the most difficult part of the problem is identifying impacted areas. Having performed this impact analysis, maintenance can proceed on only those areas affected. Micro-level rederivation strategies will allow us to limit those areas even further. The replay delta calculus approach defines the possible impacts of a maintenance delta.

The KBSA style of program development via rederivation is different than today's code-based maintenance approach. This style will pose new requirements on software development tools, a replay capability being just one of those requirements. Another area to be addressed is a new form of complexity metric beyond, for example, the McCabe metrics [17], to assess

program complexity. The replay delta calculus may permit the assessment of the modifiability of a derivation history which could serve as just the metric required, measuring the modifiability of the program via the modifiability of its design.

### Acknowledgments

This work has been supported by Rome Lab under the umbrella of the Knowledge-Based Software Assistant project. Thanks to Doug Smith and Tom Pressburger for showing us the KIDS replay facility and commenting on this work.

### References

- [1] Green, C., et al. Report on a knowledge-based software assistant. Technical Report RADC-TR-83-195, Kestrel Institute, 1983.
- [2] Rowley, S., Shrobe, H., and Cassels, R. Joshua: Uniform access to heterogeneous knowledge structures. In *6th National Conference on Artificial Intelligence*, 1987.
- [3] Kambhampati, S. A theory of plan modification. In *8th National Conference on Artificial Intelligence*, 1990.
- [4] Baxter, I. Transformational maintenance by reuse of design histories. Technical Report TR-90-36, UC Irvine, November 1990.
- [5] Goldberg, A. Reusing software developments. In *4th Annual Knowledge-Based Software Assistant Conference*, 1989.
- [6] Mostow, J. Design by derivational analogy: Issues in the automated replay of design plans. *AI Journal*, 40(1-3), September 1989.
- [7] Balzer, R. Transformational implementation: An example. *IEEE Transactions on Software Engineering*, 7(1), 1981.
- [8] Chase, M., Reubenstein, H. and Yeh, A. Incremental rederivation of software artifacts: FY 92. RL-TR 93-60, Rome Laboratory (C3CA), 525 Brooks Road, Griffiss Air Force Base, NY 13441, May 1993.
- [9] Neighbors, J. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5), 1984.
- [10] Bhansali, S and Harandi, M. The role of derivational analogy in reusing program design. In *5th Annual Knowledge-Based Software Assistant Conference*, 1990.
- [11] Kambhampati, S. Mapping and retrieval during plan reuse: A validation structure based approach. In *8th National Conference on Artificial Intelligence*, 1990.
- [12] Kambhampati, S and Hendler, J. Control of refitting during plan reuse. In *11th International Joint Conference on Artificial Intelligence*, 1989.

- [13] Blumenthal, B. Empirical comparisons of some design replay algorithms. In *8th National Conference on Artificial Intelligence*, pages 902-7, 1990.
- [14] Feather, M. Specification evolution and program (re)transformation. In *5th Annual Knowledge-Based Software Assistant Conference*, 1990.
- [15] Feather, M. Detecting interference when merging specification evolutions. In *5th International Workshop on Software Specification and Design*, 1989.
- [16] Feather, M. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2), 1989.
- [17] Gill, G and Kemerer, C. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12), December 1991.

## A Transformations Used in "A Larger Example"

This appendix gives the transformations used in the example described in Section 2.5. The transformations are given in the order that they are applied. For each transformation, the rule for that transformation is given, followed by the input to that rule and the effects of running that rule. Section 2.1 gives a description of running a sample rule to produce a transformation. Note that the system is case insensitive, so that `input` looks the same as `INPUT` and so on.

Due to implementation details, each transformation rule actually ends with a clause of the form

`& (*xformed-node* < - parm1)`

where *parm1* is the first input parameter (input in the rule fact-input below, and so on) in the rule. These clauses are not shown.

1. The rule is called fact-input. It adds input conditions to the program:

```
rule fact-input (input, name, arg)
  input = '(:input $existing-cond)' & name = 'fact' -- >
  input = '(:input (integer @(copy(arg))) (>= @(copy(arg)) 0) $existing-cond)'
```

This rule is called with the bindings `input= (:INPUT)`, `name= FACT`, and `arg= X`. The rule alters the *input* binding to be `(:INPUT (INTEGER X) (>= X 0))`.

2. The rule is called fact-output. It adds output conditions to the program:

```
rule fact-output (output, name, arg)
  output = '(:output $existing-cond)' & name = 'fact' -- >
  output = '(:output (integer (fact @(copy(arg)))) $existing cond)'
```

This rule is called with the bindings `input= (:OUTPUT)`, `name= FACT`, and `arg= X`. The rule alters the *output* binding to be `(:OUTPUT (INTEGER (FACT X)))`.

3. The rule is called fact-base. It adds the base case for finding factorials:

```
rule fact-base (implement, name, arg, design)
  implement = '(:implementation $already-imp)' & name = 'fact'
  & design = '(:design .. inductive ..)' -- >
  implement = '(:implementation
    (when (= @(copy(arg)) 0) (return-from fact 1)) $already-imp)'
```

This rule is called with the bindings `implement= (:IMPLEMENTATION)`, `name= FACT`, `arg= X`, and `design= (:DESIGN INDUCTIVE)`. The rule alters the *implement* binding to be `(:IMPLEMENTATION (WHEN (= X 0) (RETURN-FROM FACT 1)))`.

4. The rule is called fact-recur. It adds the recursive case for finding factorials:

```
rule fact-recur (implement, name, arg, design1, design2)
  implement = '(:implementation $already-imp)' & name = 'fact'
  & design1 = '(:design .. recursive ..)'
  & design2 = '(:design .. inductive ..)' -- >
  implement =
    '(:implementation
      (when (> @(copy(arg)) 0) (return-from fact
        (* @(copy(arg)) (fact (- @(copy(arg)) 1)))))
      $already-imp)'
```

This rule is called with the bindings `name= FACT`, `arg= X`, `design1= (:DESIGN RECURSIVE)`, `design2= (:DESIGN INDUCTIVE)`, and `implement=`

```
(:IMPLEMENTATION (WHEN (= X 0) (RETURN-FROM FACT 1)))
```

The rule alters the *implement* binding to be

```
(:IMPLEMENTATION (WHEN (> X 0) (RETURN-FROM FACT (* X (FACT (- X 1)))))
(WHEN (= X 0) (RETURN-FROM FACT 1)))
```

5. The rule is called test-input-conditions. It adds "code" to test the given input conditions:

```
rule test-input-conditions (implement, name, input-cond)
  implement = '(:implementation $already-imp)' & lisp-atom(name)
  & input-cond = '(:input $i-c)' -- >
  implement = '(:implementation
    (when (not (and $(copy-sequence(i-c))))
      (return-from @(copy(name)) :error)) $already-imp)'
```

This rule is called with the bindings `name= FACT`, `input-cond=`

```
(:INPUT (INTEGER X) (>= X 0))
```

and implement=

```
(:IMPLEMENTATION (WHEN (> X 0) (RETURN-FROM FACT (* X (FACT (- X 1)))))  
(WHEN (= X 0) (RETURN-FROM FACT 1)))
```

The rule alters the *implement* binding to be

```
(:IMPLEMENTATION  
(WHEN (NOT (AND (INTEGER X) (>= X 0))) (RETURN-FROM FACT :ERROR))  
(WHEN (> X 0) (RETURN-FROM FACT (* X (FACT (- X 1)))))  
(WHEN (= X 0) (RETURN-FROM FACT 1)))
```

6. The rule is called add-cache. It adds the data structures and procedures to cache a function's value:

```
rule add-cache (top-level, name, design)  
  top-level = '@existing-form1 $existing-forms'  
  & lisp-atom(name) & design = '(:design .. cache-results ..)'  
  -- >  
  str-value(cache-doc) =  
    concat("cache for ", symbol-to-string(atom-value(name))) &  
  atom-value(cache-name) =  
    gentemp(concat("*", symbol-to-string(atom-value(name)),  
                  "-CACHE*-"), "REPLAY")  
  & string-const(cache-doc) & lisp-atom(cache-name) &  
  str-value(cache-check-doc) =  
    concat("check to see if value cached for ",  
          symbol-to-string(atom-value(name))) &  
  atom-value(cache-check-name) =  
    gentemp(concat(symbol-to-string(atom-value(name)),  
                  "-VAL-CACHED?-"), "REPLAY")  
  & string-const(cache-check-doc) & lisp-atom(cache-check-name) &  
  str-value(cache-val-doc) =  
    concat("return cached value for ",  
          symbol-to-string(atom-value(name))) &  
  atom-value(cache-val-name) =  
    gentemp(concat(symbol-to-string(atom-value(name)),  
                  "-CACHED-VAL-"), "REPLAY")  
  & string-const(cache-val-doc) & lisp-atom(cache-val-name) &  
  str-value(cache-store-doc) =  
    concat("cache a value for ", symbol-to-string(atom-value(name))) &  
  atom-value(cache-store-name) =  
    gentemp(concat("CACHE-", symbol-to-string(atom-value(name))),
```

```

"-VAL-", "REPLAY")
& string-const(cache-store-doc) & lisp-atom(cache-store-name) &
top-level =
  '@existing-form1
    (defvar @cache-name (make-hash-table) @cache-doc)
    (defun @cache-check-name (x) @cache-check-doc
      (gethash x @(copy(cache-name))))
    (defun @cache-val-name (x) @cache-val-doc
      (gethash x @(copy(cache-name))))
    (defun @cache-store-name (x val) @cache-store-doc
      (setf (gethash x @(copy(cache-name))) val))
  $existing-forms'

```

This rule is called with the bindings name= FACT, design = (:DESIGN CACHE-RESULTS), and top-level=

```

(DEFUN FACT (X)
  (:DESIGN RECURSIVE) (:DESIGN CACHE-RESULTS) (:DESIGN INDUCTIVE)
  (:INPUT (INTEGER X) (>= X 0))
  (:OUTPUT (INTEGER (FACT X)))
  (:IMPLEMENTATION
    (WHEN (NOT (AND (INTEGER X) (>= X 0))) (RETURN-FROM FACT :ERROR))
    (WHEN (> X 0) (RETURN-FROM FACT (* X (FACT (- X 1)))))
    (WHEN (= X 0) (RETURN-FROM FACT 1)))
  (:COMMENT))

```

The function *gentemp* is used to insure new symbol names for generated symbols. The rule alters the *top-level* binding to be

```

(DEFUN FACT (X)
  (:DESIGN RECURSIVE) (:DESIGN CACHE-RESULTS) (:DESIGN INDUCTIVE)
  (:INPUT (INTEGER X) (>= X 0))
  (:OUTPUT (INTEGER (FACT X)))
  (:IMPLEMENTATION
    (WHEN (NOT (AND (INTEGER X) (>= X 0))) (RETURN-FROM FACT :ERROR))
    (WHEN (> X 0) (RETURN-FROM FACT (* X (FACT (- X 1)))))
    (WHEN (= X 0) (RETURN-FROM FACT 1)))
  (:COMMENT))
(DEFVAR *FACT-CACHE*-7375 (MAKE-HASH-TABLE) "cache for FACT")
(DEFUN FACT-VAL-CACHED?-7375 (X) "check to see if value cached for FACT"
  (GETHASH X *FACT-CACHE*-7375))
(DEFUN FACT-CACHED-VAL-7375 (X) "return cached value for FACT"
  (GETHASH X *FACT-CACHE*-7375))
(DEFUN CACHE-FACT-VAL-7375 (X VAL) "cache a value for FACT"
  (SETF (GETHASH X *FACT-CACHE*-7375) VAL))

```

7. The rule is called add-cache-look-up. It performs 3 actions. The first action is to check that the implementation part of a program are clauses in a certain format. The second action is to alter that format, and the third action is to have the program look for cached values before computing a new value:

```
rule add-cache-look-up (implement, name, arg, design, cache-check, cache-value)
  implement = '(:implementation $already-imp)' & lisp-atom(name)
  & (fa (clause) (clause in already-imp =>
    (clause = '(when @@ .. (return-from @@ @@))'
    & term-equal?(elements(last(elements(clause)))(2), name))))
  & design = '(:design .. cache-results ..)'
  & cache-check = '(defun @cache-check-name @@ @cache-check-doc ...)'
  & lisp-atom(cache-check-name) & string-const(cache-check-doc)
  & str-value(cache-check-doc) =
    concat("check to see if value cached for ",
    symbol-to-string(atom-value(name)))
  & cache-value = '(defun @cache-value-name @@ @cache-value-doc ...)'
  & lisp-atom(cache-value-name) & string-const(cache-value-doc)
  & str-value(cache-value-doc) =
    concat("return cached value for ",
    symbol-to-string(atom-value(name)))
  -- >
  (enumerate clause over already-imp do
    clause = '(when $main-body (return-from @@ @result))'
    -- >
    clause = '($main-body @result)')
  & implement =
    '(:implementation (cond ((@ (copy(cache-check-name)) @ (copy(arg)))
    (@ (copy(cache-value-name)) @ (copy(arg))))
    $already-imp))'
```

This rule is called with the bindings name= FACT, arg= X,

```
design= (:DESIGN CACHE-RESULTS)
cache-check=
  (DEFUN FACT-VAL-CACHED?-7375 (X)
    "check to see if value cached for FACT"
    (GETHASH X *FACT-CACHE*-7375))
cache-value=
  (DEFUN FACT-CACHED-VAL-7375 (X) "return cached value for FACT"
    (GETHASH X *FACT-CACHE*-7375))

and implement=

(:IMPLEMENTATION
```

```

(WHEN (NOT (AND (INTEGER X) (>= X 0))) (RETURN-FROM FACT :ERROR))
(WHEN (> X 0) (RETURN-FROM FACT (* X (FACT (- X 1)))))
(WHEN (= X 0) (RETURN-FROM FACT 1)))

```

The rule alters the *implement* binding to be

```

(:IMPLEMENTATION
 (COND ((FACT-VAL-CACHED?-7375 X) (FACT-CACHED-VAL-7375 X))
       ((NOT (AND (INTEGER X) (>= X 0))) :ERROR)
       ((> X 0) (* X (FACT (- X 1)))))
       ((= X 0) 1)))

```

8. The rule is called add-cache-save-out. It alters a program to save into the cache the results of any computation:

```

rule add-cache-save-out (implement, name, arg, design, cache-save-fcn)
  implement = '(:implementation (cond $cond-clauses))'
  & lisp-atom(name) & design = '(:design .. cache-results ..)'
  & cache-save-fcn = '(defun @cache-save-name @@ @cache-save-doc ...)'
  & lisp-atom(cache-save-name) & string-const(cache-save-doc)
  & str-value(cache-save-doc) = concat("cache a value for ",
                                       symbol-to-string(atom-value(name)))
  -- >
  atom-value(ret-var) = gentemp("RET-", "REPLAY") & lisp-atom(ret-var) &
  implement = '(:implementation
    (let ((@ret-var (cond $cond-clauses)))
      (@(copy(cache-save-name)) @(copy(arg)) @(copy(ret-var)))
      @(copy(ret-var))))'

```

This rule is called with the bindings name= FACT, arg= X, design= (:design cache-results), cache-save-fcn=

```

(DEFUN CACHE-FACT-VAL-7375 (X VAL) "cache a value for FACT"
  (SETF (GETHASH X *FACT-CACHE*-7375) VAL))

```

and implement=

```

(:IMPLEMENTATION
 (COND ((FACT-VAL-CACHED?-7375 X) (FACT-CACHED-VAL-7375 X))
       ((NOT (AND (INTEGER X) (>= X 0))) :ERROR)
       ((> X 0) (* X (FACT (- X 1)))))
       ((= X 0) 1)))

```

The rule alters the *implement* binding to be



```

(:IMPLEMENTATION
(LET ((RET-7375
      (COND ((FACT-VAL-CACHED?-7375 X) (FACT-CACHED-VAL-7375 X))
            ((NOT (AND (INTEGER X) (>= X 0))) :ERROR)
            ((> X 0) (* X (FACT (- X 1))))
            ((= X 0) 1))))
      (CACHE-FACT-VAL-7375 X RET-7375)
      RET-7375))

```

***MISSION  
OF  
ROME LABORATORY***

**Mission.** The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.